



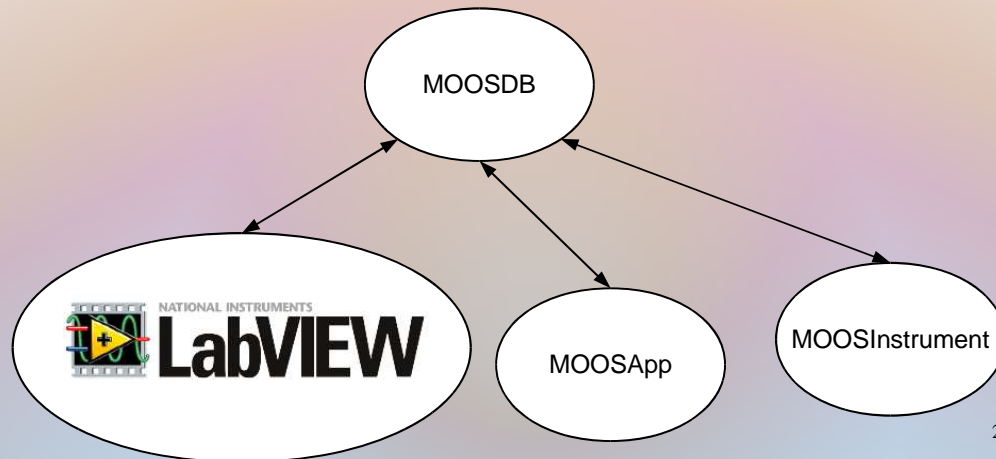
STEVENS
INSTITUTE *of* TECHNOLOGY
THE INNOVATION UNIVERSITY

Native LabVIEW MOOS interface

Alexander Sedunov

Native LabVIEW MOOS interface

- MOOS communication library based only on built-in Labview functions



We have developed a communication framework by replicating the essential functionality of MOOSLib class CMOOSCommClient in Labview environment, relying solely on built-in functions.

Presentation outline

- Why use LabVIEW?
- Labview interface
- Why native implementation?
- Replicating CMOOSCommClient
- Maintaining the interface
- Demonstration

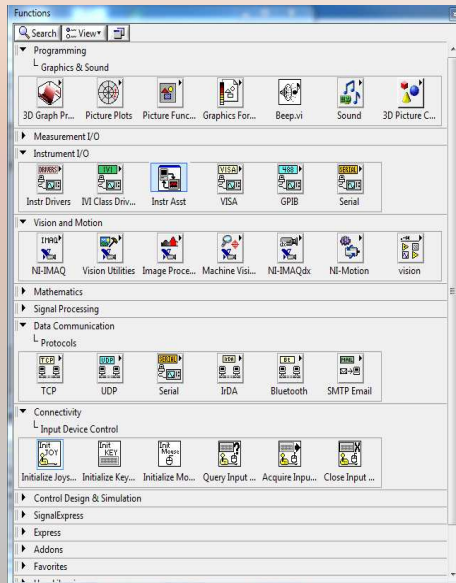


We will talk about

- Why it is desirable to have the functionality of Labview
- Why we took this implementation approach
- What it will take to maintain this interface
- And will show one of our internal tools which is based on this framework

Why use NI LabVIEW

- Rapid prototyping
- Hardware interfacing
- Instant portability to supported platforms

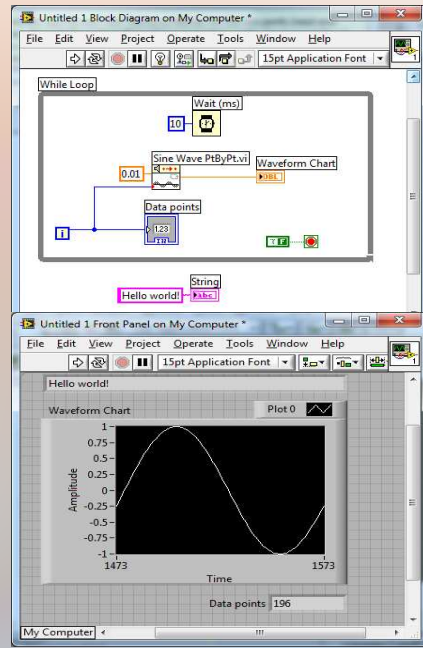


4

National Instruments LabVIEW™ has gained popularity as a platform for engineers and scientists and integrating it with MOOS allows for the easy development of graphical MOOS utilities in an environment having access to vast libraries for control, communication, data acquisition, signal processing and ability to run on many platforms including Windows, Linux, Mac OS X.

Virtual Instruments (VI)

- Programs in Labview are called "Virtual Instruments"
- Consist of diagram and Front Panels



The concept of programming in Labview is quite unconventional. Labview programs are called "Virtual Instruments".

They consist of diagrams and front panels. The diagrams contain the actual programs, in graphical language "G" and front panels contain the user interface.

The Native MOOS Interface API

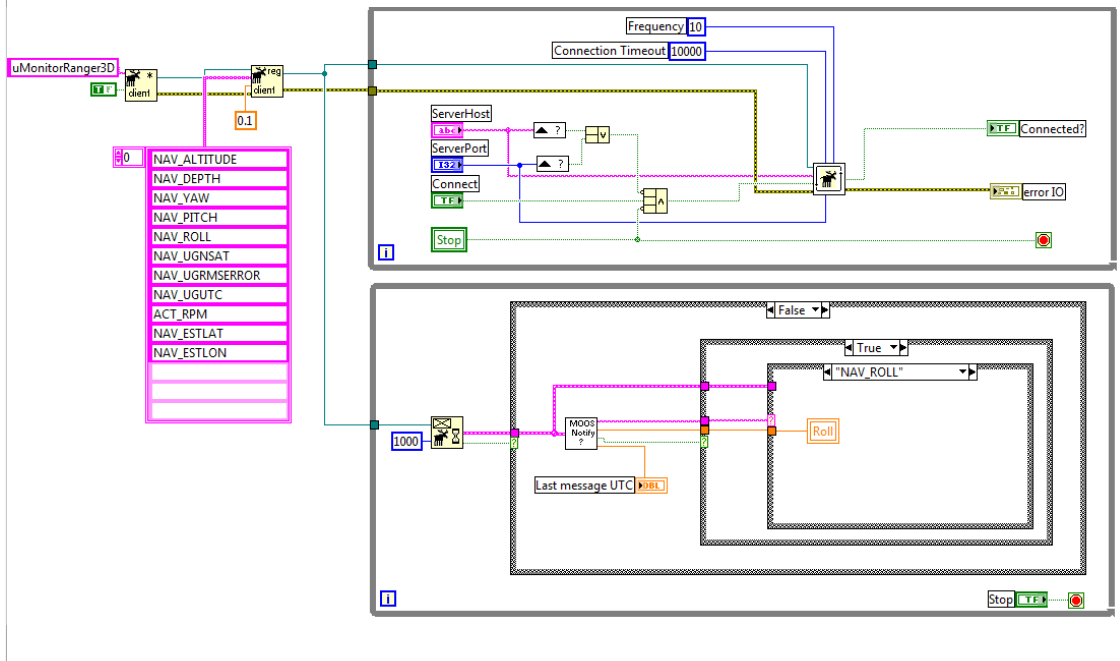
→ Click to add an outline



6

We have created an MOOS API for Labview, which consists of only 6 Vis, which need to be included in a typical application to exchange data with MOOSDB.

Sample LabVIEW diagram

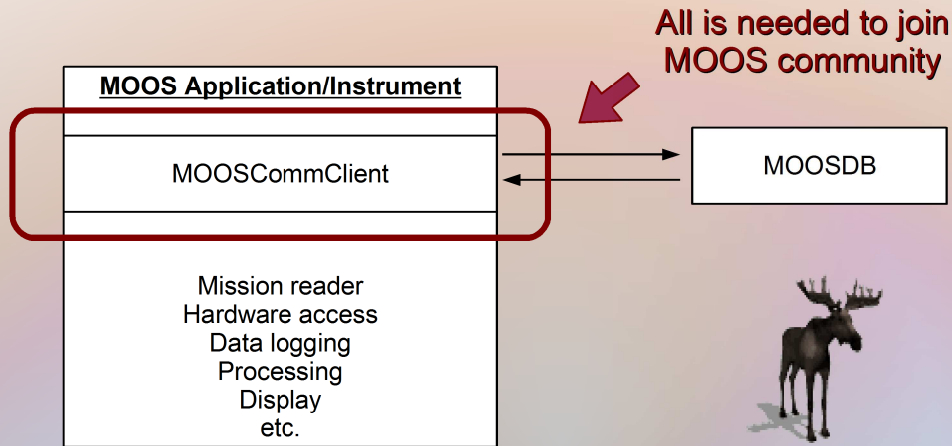


You can see a diagram of a skeletal application which subscribes to multiple MOOS variables and updates corresponding variables in Labview.

As you can see it basically consists of initializing the client, giving it the list of variables for it to subscribe and two loops, which in Labview imply two separate threads. One is over a function which performs all the client work, including connection and handshake, the other loop is merely a consumer of messages generated by the first, sorting the notifications by key into appropriate variables.

MOOS Community

→ Click to add an outline



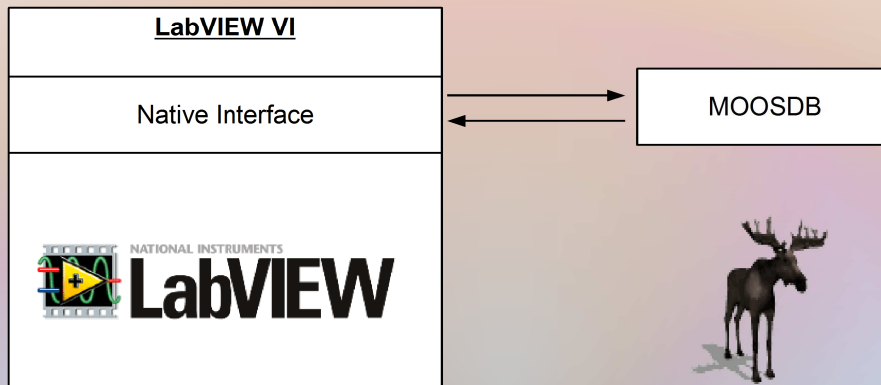
8

MOOS libraries provide many tool:, there is infrastructure for most tasks of a robotic platform.

However the external communication of a MOOS application is confined to a single class: the **MOOSCommClient**, which allows it to join a MOOS community and share data through **MOOSDB**.

Native LabVIEW MOOS Interface

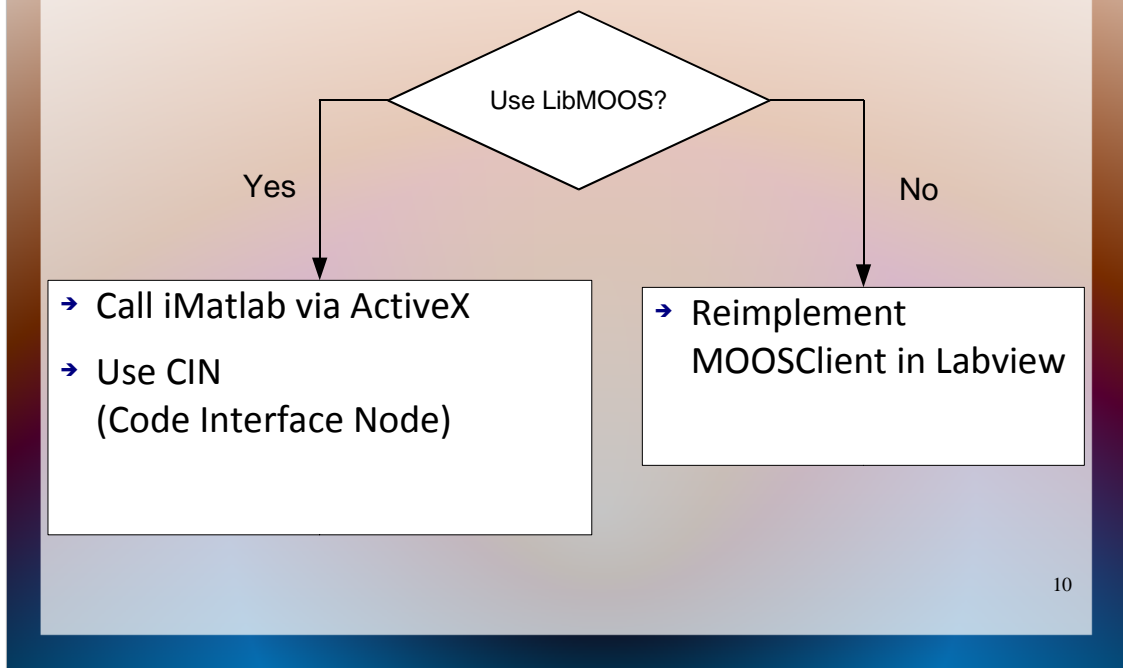
→ Click to add an outline



9

Our goal is to allow Labview applications to join a MOOS community. The bare minimum requirement for that is to provide it with some implementation of CMOOSCommClient.

Alternative ways of integration



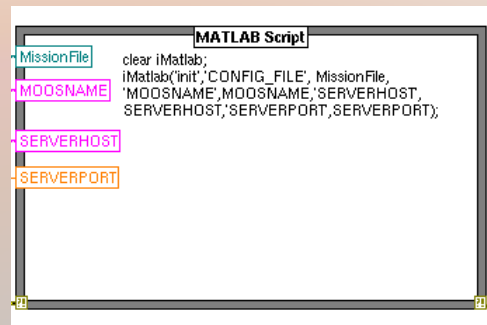
When we approached the problem we had two ways of proceeding. We could make some interface for MOOS libraries to Labview, having the advantage that if any new development will happen which will change the interface we could just recompile to integrate with future versions of MOOS.

Another way is to replicate the protocol using the Labview functionality.

Further we consider those two avenues in detail.

Call iMatlab via ActiveX

- ActiveX is Windows only
- Considerable overhead
- Only one instance per computer!



```
MATLAB Script
clear iMatlab;
iMatlab('init','CONFIG_FILE', MissionFile,
'MOOSNAME',MOOSNAME,'SERVERHOST',
SERVERHOST,'SERVERPORT',SERVERPORT);
```

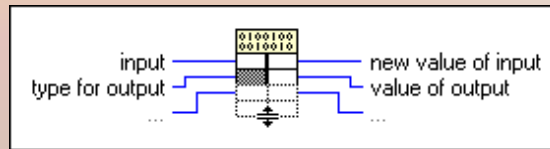
11

If both Matlab and Labview are available on the same computer it is easy to use the pre-existing iMatlab, a MOOS instrument compiled as a MATLAB Executable (MEX), which can be run using Labview's "Matlab Script Node", a mechanism based on connecting to ActiveX server provided by Matlab. This has notable limitations: iMatlab allows to initialize only one global instance per computer, meaning that all applications will have only one connection to single MOOSDB and if another client tries to initialize there will be a conflict. The functionality is only available on Windows plus the overhead to exchange data between Labview and Matlab is very considerable, making this solution applicable only for uses not requiring high performance.

Code Interface Node

- Single entry point
- Callbacks for loading and unloading VI
- Everything has to be encapsulated into single function

Code Interface Node



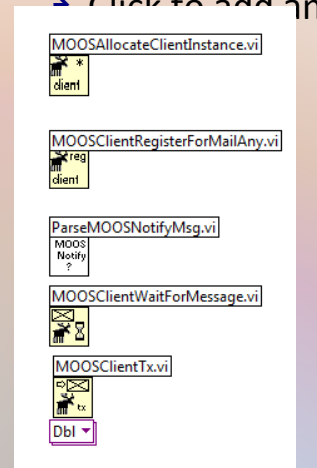
12

Labview provides many ways to call external code in shared libraries, ActiveX objects and using a Code Interface Node (CIN). The latter is in essence a special format of compiled code which can be embedded into Labview as a VI, it has several special entry points which are called whenever certain events, such as aborting execution by user occur, enabling to perform "housekeeping" necessary. CIN similarly to MEX can implement only a single function which will have to wrap all the functionality, so while it is possible create an interface, it will be impractical to manage more than one instance of a MOOSClient thus imposing same limitations as iMatlab.

Labview also able to use ActiveX and creating a wrapper with this technology can be seen as way to maintain multiple instances, but this automatically limits us to only using Windows platform.

Reimplement MOOSClient in Labview

- Resources managed automatically
- Unlimited instances per machine
- Low-level Vis exposed

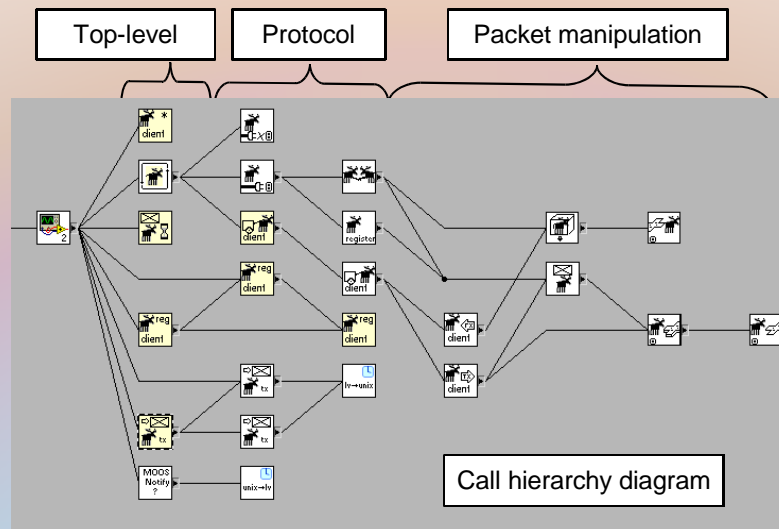


13

The last option is to use the built-in functions of Labview to mimic the protocol used by MOOS. As a result the Labview program can be seamlessly transferred between all platforms which are supported. By relying on Just-In-Time native code compilation Labview achieves high performance, so the processing overhead introduced is minimal. There is a very developed infrastructure in Labview for management of resources such as sockets and memory as well for concurrent execution and managing instances of virtual instruments, making it easy to incorporate several MOOS instruments into one application. A separate package called "Application Builder" enables to create executable files which can be run on any computer where Labview Runtime can be installed, the implementation we provide relies only on "Base Package" functionality and thus requires only minimal runtime with no additional packages.




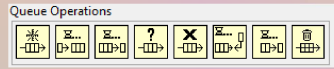
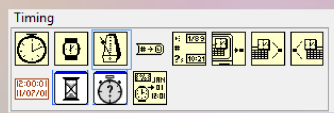
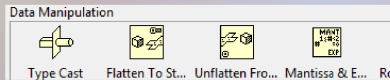
Reimplement MOOSClient in Labview

Low-level VIs exposed to developer

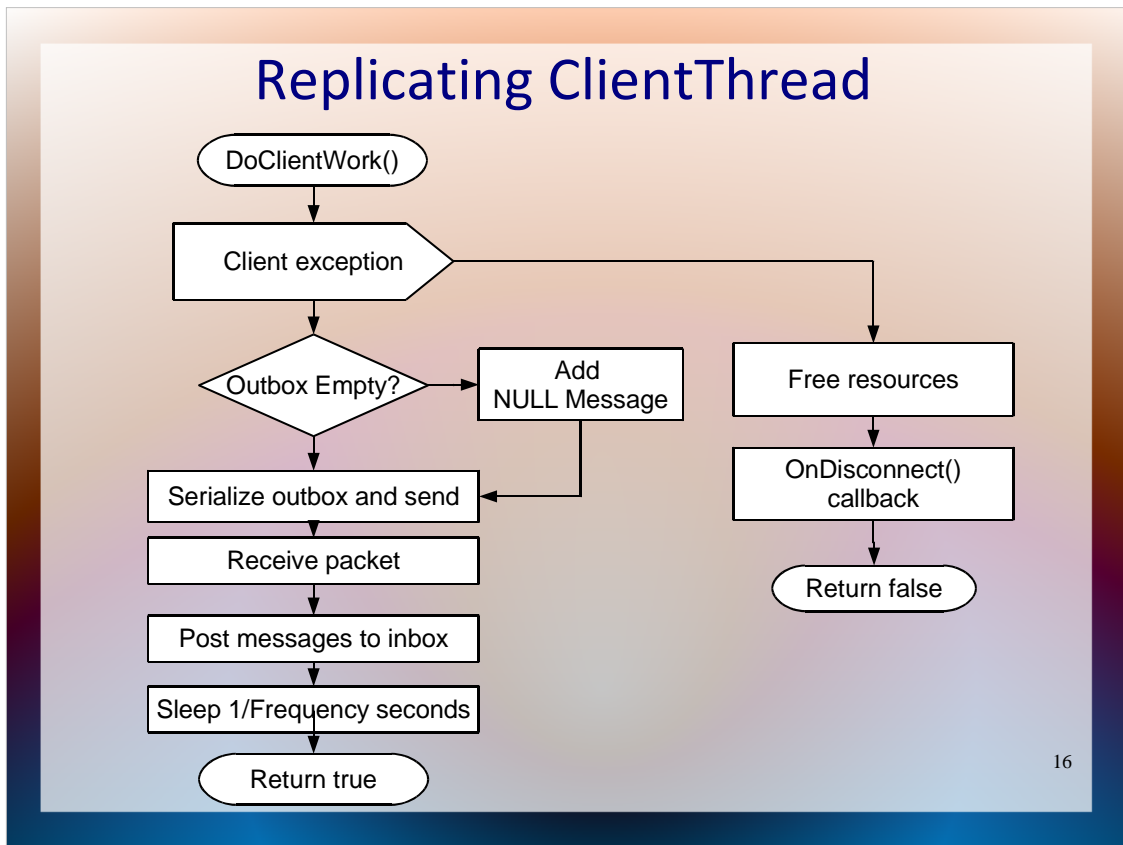


Another important benefit is that such implementation automatically exposes all the intermediate low-level functions for MOOS communication. This way we have more flexibility than wrapping high-level MOOS classes would allow, for example the functions used to serialize MOOS messages can be easily reused to forward MOOS data structures via different protocols such as UDP or even entirely different communication media than Internet.

LibMOOS vs LabVIEW functionality

Function		
Threads	MOOSThread.h	Implicit parallelism
Sockets	XPCSocket.h	
Queues	MOOSMSG_LIST	
Time	MOOSGenLibGlobal Helper.h	
Serialization	MOOSMsg.h	

To have a better idea, why such reimplementation is a viable solution, let us take a look at the functionality provided by MOOS. Many features are cross-platform implementations of such functions as threads, network sockets, serialization, time it also uses some C++ STL data structures which are needed to reimplement it. Now if we take a look at the functionality Labview provides, we can see what most of this infrastructure is already there, so there is no effort involved in providing most of it to a program mimicking the protocol.



What is left is to recreate the functionality of CMOOSCommClient, which at high level is not very complicated.

Maintaining separate implementation of interface

- Protocol went unchanged for long
- Implementation is confined to few functions

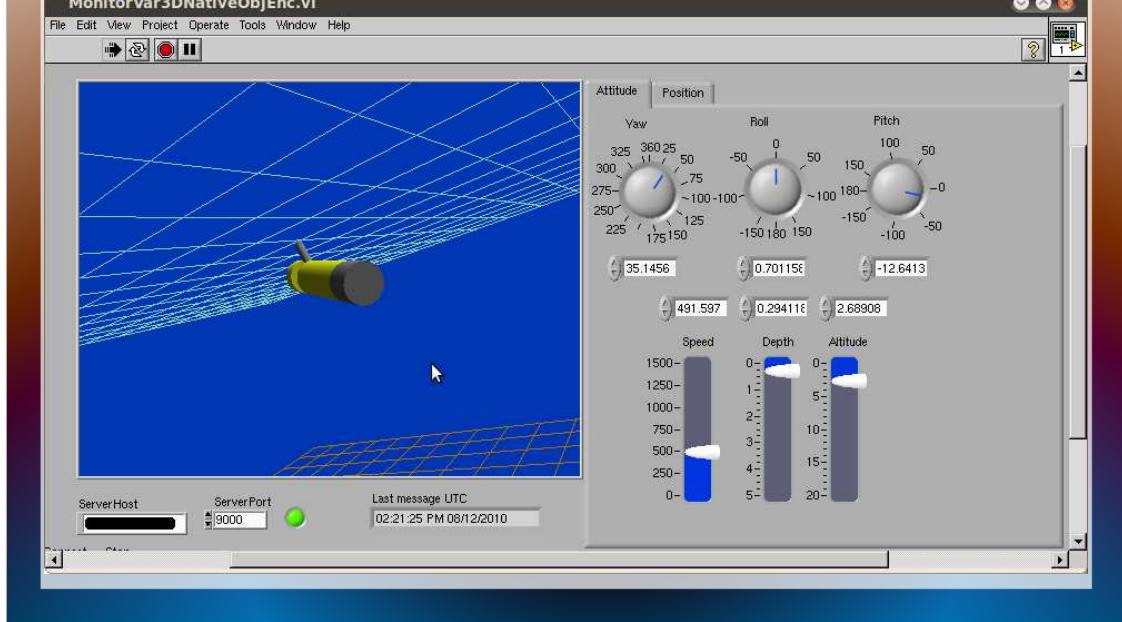
Files to monitor between versions:

- MOOSCommClient.cpp/.h
- MOOSCommObject.cpp/.h
- MOOSCommPkt.cpp/.h
- MOOSMsg.cpp/.h

17

A trade-off of this method is that is that the implementation will have to be updated separately with any changes to the MOOS communication protocol, but availability of source code makes it a simple task. The minimalistic approach for integration minimizes the dependency on the changes in modules of MOOS. There are just a few functions replicated from the total of 4 source files, which with any update will have to be compared to the versions on which the implementation is based.

Demo: LabVIEW-based vehicle attitude display



Labview makes it easy to develop graphical tools. For example it was easy for us to develop a tool which monitors the attitude of a vehicle during a mission. With it we could easily analyse asynchronous logs produced by pLogger by simply looking at 3D model of the vehicle instead of a set of graphs.

Acknowledgement

- This work was supported by ONR project #N00014-05-1-0632: Navy Force Protection Technology Assessment Project
- This work would have been impossible without the efforts of Paul Newman and all the contributors to MOOS development.